

BliStrTune: Hierarchical Invention of Theorem Proving Strategies

Jan Jakubův Josef Urban

Czech Technical University, CIIRC, Prague
{jakubuv,josef.urban}@gmail.com

Abstract

Inventing targeted proof search strategies for specific problem sets is a difficult task. State-of-the-art automated theorem provers (ATPs) such as E allow a large number of user-specified proof search strategies described in a rich domain specific language. Several machine learning methods that invent strategies automatically for ATPs were proposed previously. One of them is the Blind Strategymaker (BliStr), a system for automated invention of ATP strategies.

In this paper we introduce BliStrTune – a hierarchical extension of BliStr. BliStrTune allows exploring much larger space of E strategies by interleaving search for high-level parameters with their fine-tuning. We use BliStrTune to invent new strategies based also on new clause weight functions targeted at problems from large ITP libraries. We show that the new strategies significantly improve E’s performance in solving problems from the Mizar Mathematical Library.

Keywords Automated Theorem Proving, Machine Learning, Proof Search Heuristics, Clause Weight Functions

1. Introduction: ATP Strategy Invention

State-of-the-art automated theorem provers (ATPs) such as E (Schulz 2002, 2013) and Vampire (Kovács and Voronkov 2013) achieve their performance by using sophisticated proof search strategies and their combinations. Constructing good ATP search strategies is a hard task that is potentially very rewarding. Until recently, there has been, however, little research in this direction in the ATP community.

With the arrival of large ATP problem sets and benchmarks extracted from the libraries of today’s interactive theorem prover (ITP) systems (Blanchette et al. 2016a,b; Gauthier and Kaliszyk 2015; Kaliszyk and Urban 2014, 2015), automated generation of targeted ATP strategies became an attractive topic. It seems unlikely that manual (“theory-driven”) construction of targeted strategies can scale to large numbers of ATP problems spanning many different areas of mathematics and computer science. Starting with Blind Strategymaker (BliStr) (Urban 2015) that was used to invent E’s strategies for MaLAREa (Urban et al. 2008; Kaliszyk et al. 2015b) on the 2012 Mizar@Turing competition prob-

lems (Sutcliffe 2013), several systems have been recently developed to invent targeted ATP strategies (Schäfer and Schulz 2015; Kühlwein and Urban 2015). The underlying methods used so far include genetic algorithms and iterated local search, as popularized by the ParamILS (Hutter et al. 2009) system.

A particular problem of the methods based on iterated local search is that their performance degrades as the number of possible strategy parameters gets high. This is the case for E, where a domain specific language allows construction of astronomic numbers of strategies. This gets worse as more and more sophisticated templates for strategies are added to E, such as our recent family of conjecture-oriented weight functions implementing various notions of term-based similarity (Jakubův and Urban 2016). The pragmatic solution used in the original BliStr consisted of re-using manually pre-designed high-level strategy components, rather than allowing the system to explore the space of all possible strategies. This is obviously unsatisfactory.

In this work we introduce BliStrTune – a hierarchical extension of BliStr. BliStrTune allows exploring much larger space of E strategies by factoring the search into invention of good high-level strategy components and their low-level fine-tuning. The high-level and low-level inventions communicate to each other their best solutions, iteratively improving all parts of the strategy space. Together with our new conjecture-oriented weight functions, the hierarchical invention produces so far the strongest schedule of strategies on the small (*bushy*) versions of the Mizar@Turing problems. The improvement over Vampire 4.0 on the training set is nearly 10%, while the improvement on the testing (competition) set is over 5%.

The rest of the paper is organized as follows. Section 2 introduces the notion of proof search strategies, focusing on resolution/superposition ATPs and E prover. We also summarize our recent conjecture-oriented strategies that motivated the work on BliStrTune. Section 3 describes the ideas behind the original Blind Strategymaker based on the ParamILS system (see Section 3.1 for more details on ParamILS). Section 4 introduces the hierarchical invention algorithm and its implementation. The system is evaluated

in several ways in Section 5, showing significant improvements over the original BliStr and producing significantly improved ATP strategies.

2. Proof Search Strategies

In this section we briefly describe the proof search of saturation-based automated theorem provers (ATPs). Section 2.1 describes the proof search control possibilities of E prover (Schulz 2002, 2013). Section 2.2 describes our previous development of similarity based clause selection strategies (Jakubův and Urban 2016) which we make use of and evaluate here.

Many state-of-the-art ATPs are based on the *given clause algorithm* introduced by Otter (McCune 1989, 1990, 1994). The input problem $T \cup \{\neg C\}$ is translated into a refutationally equivalent set of clauses. Then the search for a contradiction, represented by the empty clause, is performed maintaining two sets: the set P of *processed clauses* and the set U of *unprocessed clauses*. Initially, all the input clauses are unprocessed. The algorithm repeatedly selects a *given clause* g from U and generates all possible inferences using g and the processed clauses from P . Then, g is moved to P , and U is extended with the newly produced clauses. This process continues until a resource limit is reached, or the empty clause is inferred, or P becomes *saturated*, that is, nothing new can be inferred.

2.1 Proof Search Strategies in E Prover

E (Schulz 2002, 2013) is a state-of-the-art theorem prover which we use as a basis for implementation. The selection of a given clause in E is implemented by a combination of priority and weight functions. A *priority function* assigns an integer to a clause and is used to pre-order clauses for weight evaluation. A *weight function* takes additional specific arguments and assigns to each clause a real number called *weight*. A *clause evaluation function* (CEF) is specified by a priority function, weight function, and its arguments. Each CEF selects the clause with the smallest pair (*priority*, *weight*) for inferences. Each CEF is specified using the syntax

WeightFunction(PriorityFunction,...)

with a variable number of comma separated arguments of the weight function. E allows a user to select an *expert heuristic* on a command line in the format

(n_1 *CEF₁, ..., n_k *CEF_k)

where integer n_i indicates how often the corresponding CEF _{i} should be used to select the given clause. E additionally supports an *auto-schedule* mode where several expert heuristics are tried, each for a selected time period. The heuristics and time periods are automatically chosen based on input problem properties.

One of the well-performing weight functions in E, which we also use as a reference for evaluation of our weight functions, is the *conjecture symbol weight*. This weight function counts symbol occurrences with different weights based on their appearance in the conjecture as follows. Different weights δ_f , δ_c , δ_p , and δ_v are assigned to function, constant, and predicate symbols, and to variables. The weight of a symbol which appears in the conjecture is multiplied by γ_{conj} , typically $\gamma_{\text{conj}} < 1$ to prefer clauses with conjecture symbols. To compute a term weight, the given symbol weights are summed for all symbol occurrences. This evaluation is extended to equations and to clauses.

Apart from clause selection, E prover introduces other parameters which influence the choice of the inference rules, term orderings, literal selection, etc. The selected values of the parameters which control the proof search are called a *protocol*. Because *protocol* is a crucial notion in this paper, we provide a simple example for reader's convenience.

EXAMPLE 1. *Let us consider the following simplified E protocol written in E prover command line syntax as follows.*

```
-tKB06 -WSelectComplexG
-H' (13*Refinedweight(PreferGoals,1,2,2,3,2),
    2*Clauseweight(ByCreationDate,-2,-1,0.5))'
```

This protocol selects term ordering KB06, literal selection function SelectComplexG, and two CEFs. The first CEF has frequency 13, weight function Refinedweight, priority function PreferGoals, and weight function arguments "1,2,2,3,2". An exact meaning of specific protocol parameters can be found in E manual (Schulz 2013).

2.2 Similarity Based Clause Selection Strategies

Many of the best-performing weight functions in E are based on a similarity of a clause with the conjecture, for example, the *conjecture symbol weight* from the previous section. A natural question arises whether or not it makes sense to extend the symbol-based similarity to more complex term-based similarities. Previously we proposed (Jakubův and Urban 2016), implemented, and evaluated several weight functions which utilize conjecture similarity in different ways. Typically they extend the symbol-based similarity by similarity on terms. Using finer formula features improves the high-level premise selection task (Kaliszyk et al. 2015a), which motivated us on steering also the internal selection in E. The following sections summarize the new weight functions which we further evaluate later in Section 5.1 and Section 5.3.

2.2.1 Conjecture Subterm Weight (Term)

The first of our weight functions is similar to the standard *conjecture symbol weight*, counting instead of symbols the number of subterms a term shares with the conjecture. The clause weight function Term takes five specific arguments γ_{conj} , δ_f , δ_c , δ_p and δ_v . The weight of a term equals weight δ_f for functional terms, δ_c for constants, δ_p for predicates, and

δ_v for variables, possibly multiplied by γ_{conj} when t appears in the conjecture. To compute a clause weight, terms weights are summed for all subterms from a clause.

2.2.2 Conjecture Frequency Weight (Tfldf)

Term frequency – inverse document frequency, is a numerical statistic intended to reflect how important a word is to a document in a corpus (Leskovec et al. 2014). A *term frequency* is the number of occurrences of the term in a given document. A *document frequency* is the number of documents in a corpus which contain the term. The term frequency is typically multiplied by the logarithm of the inverse of document frequency to reduce frequency of terms which appear often. We define $\text{tf}(t)$ as the number of occurrences of t in a conjecture. We consider a fixed set of clauses denoted Docs. We define $\text{df}(t)$ as the count of clauses from Docs which contain t . Our weight function Tfldf takes one specific argument δ_{doc} to select documents, either (1) ax for the axioms (including the conjecture) or (2) pro for all the processed clauses. First we define the value $\text{tfldf}(t)$ of term t as follows.

$$\text{tfldf}(t) = \text{tf}(t) * \log \frac{1 + |\text{Docs}|}{1 + \text{df}(t)}$$

The weight of term t is computed as $\frac{1}{1 + \text{tfldf}(t)}$ and extended to clauses.

2.2.3 Conjecture Term Prefix Weight (Pref)

The previous weight functions rely on an exact match of a term with a conjecture related term. The following weight function loosen this restriction and consider also partial matches. We consider terms as symbol sequences. Let $\text{max-pref}(t)$ be the longest prefix t shares with a conjecture term. A *term prefix weight* (Pref) counts the length of $\text{max-pref}(t)$ using weight arguments δ_{match} and δ_{miss} . These are used to define the weight of term t as follows.

$$\delta_{\text{match}} * |\text{max-pref}(t)| + \delta_{\text{miss}} * (|t| - |\text{max-pref}(t)|)$$

2.2.4 Conjecture Levenshtein Distance Weight (Lev)

A straightforward extension of Pref is to employ the Levenshtein distance (Levenshtein 1966) which measures a distance of two strings as the minimum number of edit operations (character insertion, deletion, or change) required to change one word into the other. Our weight function Lev defines the weight of term t as the minimal Levenshtein distance from t to some conjecture term. It takes additional arguments δ_{ins} , δ_{del} , δ_{ch} to assign different costs for edit operations.

2.2.5 Conjecture Tree Distance Weight (Ted)

The Levenshtein distance does not respect a tree structure of terms. To achieve that, we implement the *Tree edit distance* (Zhang and Shasha 1989) which is similar to Levenshtein but uses tree editing operations (inserting a node into a tree, deleting a node while reconnecting its child nodes to the deleted position, and renaming a node label). Our weight

function Ted takes the same arguments as Lev above and term weight is defined similarly.

2.2.6 Conjecture Structural Distance Weight (Struc)

With Ted, a tree produced by the edit operations does not need to represent a valid term as the operations can change number of child nodes. To avoid this we define a simple *structural distance* which measures a distance of two terms by a number of *generalization* and *instantiation* operations. Generalization transforms an arbitrary term to a variable while instantiation does the reverse. Our weight function Struc takes additional arguments δ_{miss} , γ_{inst} , and γ_{gen} as penalties for variable mismatch and operation costs. The distance of a variable x to a term t is the cost of instantiating x by t , computed as $\Delta_{\text{Struc}}(x, t) = \gamma_{\text{inst}} * |t|$. The distance of t to x is defined similarly but with γ_{gen} . A distance of non-variable terms t and s which share the top-level symbol is the sum of distances of the corresponding arguments. Otherwise, a generic formula $\Delta_{\text{Struc}}(t, x_0) + \Delta_{\text{Struc}}(x_0, s)$ is used. The term weight is as for Lev but using Δ_{Struc} .

3. Blind Strategymaker (BliStr)

In this section we describe Blind Strategymaker (BliStr) (Urban 2015) which we further extend in the following section. BliStr is a system that develops E prover protocols targeted for a given large set of problems. The main idea is to interleave (i) iterated low-timelimit local search for new protocols on small sets of similar easy problems with (ii) higher-timelimit evaluation of the new protocols on all problems. The accumulated results of the global higher-timelimit runs are used to define and evolve the notion of similar easy problems, and to control the selection of the next protocol to be improved.

The main criterion for BliStr is as follows.

CRITERION 1 (Max). *Invent a set of E protocols that together solve as many of the given benchmark problems.*

To ensure that the invented protocols perform well also on unknown but related problems a second criterion is considered.

CRITERION 2 (Gen). *The protocols should be reasonably general.*

To simplify employment of the invented protocols, BliStr tries to achieve also the third criterion.

CRITERION 3 (Size). *The set of such protocols should not be too large.*

As defined earlier, E protocols consist of many parameters and their values which influence the proof search. A huge number of weight function arguments within clause evaluation functions (CEFs, see Section 2.1) makes the set of meaningful protocol parameters very large for a straightforward use of iterative local search as done by the ParamILS (Hutter et al. 2009) system. Since ParamILS oth-

erwise looks like the right tool for the task, a data-driven (“blind”) approach was applied in the original BliStr to get a smaller set of meaningful CEFs: the existing E protocols that were most useful on benchmarks of interest were used to extract a smaller set (a dozen) of CEFs. Making this CEFs choice more “blind” is the main contribution of this work and it is discussed in details in Section 4.

Even after such reduction, the space of the protocol parameter-value combinations is so large that a random exploration seems unlikely to find good new protocols. The guiding idea in BliStr is to use again a data-driven approach. Problems in a given mathematical field often share a lot of structure and solution methods. Mathematicians become better and better by solving the problems, they become capable of doing larger and larger steps with confidence, and as a result they can gradually attack problems that were previously too hard for them. By this analogy, it is plausible to think that if the solvable problems become much easier for an ATP system, the system will be able to solve some more (harder, but related) problems. For this to work, a method that can improve an ATP on a set of solvable problems is needed. As already mentioned, the established ParamILS system can be used for this.

3.1 ParamILS and Its Use in the BliStr Loop

Let A be an algorithm whose parameters come from a *configuration space* (product of possible values) Θ . A *parameter configuration* is an element $\theta \in \Theta$, and $A(\theta)$ denotes the algorithm A with the parameter configuration θ . Given a distribution (set) of problem instances D , the *algorithm configuration problem* is to find the parameter configuration $\theta \in \Theta$ resulting in the best performance of $A(\theta)$ on the distribution D . ParamILS is an implementation of an *iterated local search* (ILS) algorithm for the algorithm configuration problem. In short, starting with an initial configuration θ_0 , ParamILS loops between two steps: (i) perturbing the configuration to escape from a local optimum, and (ii) iterative improvement of the perturbed configuration. The result of step (ii) is accepted if it improves the previous best configuration.

To fully determine how to use ParamILS in a particular case, A , Θ , θ_0 , D , and a performance metric need to be instantiated. In our case, A is E run with a low time-limit t_{cutoff} , Θ is the set of expressible E protocols, and as a performance metric we use the number of given-clause loops done by E during solving the problem. If E cannot solve a problem within the low timelimit, a sufficiently high value (10^6) is used. Since it is unlikely that there is one best E protocol for all of the given benchmark problems, it would be counterproductive to use all problems as the set D for ParamILS runs. Instead, BliStr partitions the set of all solvable problems into subsets on which the particular protocols perform best. See (Urban 2015) for the technical details of the BliStr heuristic for choosing the successive θ_0 and D . The complete BliStr loop then iteratively co-evolves

the set of protocols, the set of solved problems, the matrix of the best results, and the set of the protocols eligible for the ParamILS improvement together with their problem sets.

4. BliStrTune: Hierarchical Invention

BliStr uses a fixed set of CEFs for inventing new protocols. The arguments of these fixed CEFs (the priority function, weight function arguments) cannot be modified during the iterative protocol improvement done by ParamILS. A straightforward way to achieve invention (fine-tuning) of CEF arguments would be to extend the ParamILS configuration space Θ . This, however, makes the configuration space grow from ca. 10^7 to 10^{120} of possible combinations. Preliminary experiments revealed that with a configuration space of this size ParamILS does not produce satisfactory results in a reasonable time.

In this section we describe our new extension of BliStr – BliStrTune – where the invention of good high-level protocol parameters (Section 4.1) is interleaved with the invention of good CEF arguments (Section 4.2). The basic idea behind BliStrTune is iterated *hierarchical invention*: The large space of the optimized parameters is naturally factored into two (in general several) layers, and at any time only one layer is subjected to invention, while the other layer(s) remain fixed. The results then propagate between the layers, and the layer-tuning and propagation are iterated. BliStrTune is experimentally evaluated in Section 5.

4.1 Global Parameter Invention

The ParamILS runs used in the BliStrTune’s global-tuning phase are essentially the same as in the case of BliStr, with the following minor exceptions. BliStr uses a fixed configuration space Θ for all ParamILS runs. This is possible because a small set (currently 12) of CEFs is hard coded in BliStr’s Θ . BliStrTune uses in the global-tuning phase a parametrized configuration space Θ_C where C is a collection of CEFs that can be different for each ParamILS run. This collection can be arbitrary but we use only the 50 best performing CEFs in order to limit the configuration space size for the global-tuning phase. The notion of “best performing CEFs” develops in time and it is discussed in details in Section 4.3. Furthermore, BliStrTune introduces additional argument c_{cef} to limit the maximum number of CEFs which can occur in a single protocol ($c_{cef} = 12$ for the case of BliStr).

BliStrTune’s global-tuning usage of ParamILS is otherwise the same as in BliStr, that is, given Θ_C , the initial configuration $\theta_0 \in \Theta_C$, and problems D , the result of the global tuning is a configuration $\theta_1 \in \Theta_C$ which has the best found performance on D . This configuration θ_1 then serves as an input for the next fine-tuning phase.

EXAMPLE 2. Let us consider the *E* protocol from Example 1. In the global-tuning phase we instruct ParamILS to modify top level arguments, that is, term ordering (“-t”), literal selection (“-W”), CEF frequencies (“13*” and “2*”), and also the whole CEF blocks and their count. We do not, however, allow ParamILS to change CEF arguments (priority functions and weight function arguments). The whole CEF must be changed to another CEF from collection *C*.

4.2 Invention of the CEF Arguments

Given the result of the global-tuning phase $\theta_1 \in \Theta_C$ a new configuration space for the fine-tuning phase Θ_{θ_1} is constructed by (1) fixing the parameter values from θ_1 and by (2) an introduction of new parameters that allow to change the values of the arguments of the CEFs used in θ_1 . In order to do that, we need to describe the space of the possible values of the CEF arguments.

The CEF arguments (see Section 2.1) consist of the priority function and the weight function specific arguments. Because of the different number and semantics of the weight function arguments, we do not allow to change the CEF’s weight functions during the fine-tuning. They are fixed to the values provided in θ_1 . For each weight function argument, we know its type (such as the symbol *weight*, operation *cost*, weight *multiplier*, etc.). For each type we have pre-designed the set of reasonable values. For the original *E* weight functions, we extract the reasonable values from the auto-schedule mode of *E*. For our new weight functions, we use our preliminary experiments (Jakubův and Urban 2016) enhanced with our intuition.

Given the configuration space Θ_{θ_1} , a configuration $\theta_1 \in \Theta_C$ can be easily converted to an equivalent configuration $\theta'_1 \in \Theta_{\theta_1}$ by setting the parameter values to those CEFs arguments that were previously fixed in θ_1 and *C*. Then we can run ParamILS with the configuration space Θ_{θ_1} , the initial configuration θ'_1 , and with the same problem set *D* as in the global-tuning phase. The result is a configuration $\theta'_2 \in \Theta_{\theta_1}$ providing the best found performance on *D*.

The global invention (global tuning) and the local invention (fine-tuning) phases can be iterated. To do that, we need to transform the result of the fine-tuning $\theta'_2 \in \Theta_{\theta_1}$ to an equivalent initial configuration $\theta_2 \in \Theta_C$ for the next global-tuning phase. In order to do that, the CEFs invented by θ'_2 must be present in the CEFs collection *C*. If this is not the case, we simply extend *C* with the new CEFs. In practice, we now use two iterations of this process (that is, two phases of global-tuning and two phases of fine-tuning) which was experimentally evaluated to provide good results.

EXAMPLE 3. Recall the protocol from Example 1 and Example 2. In the fine-tuning phase we would fix all the top level arguments modified in global-tuning phase (“-t”, and so on, as described in Example 2) and we would instruct ParamILS to change individual CEF arguments. That is, the values

```
PreferGoals,1,2,2,3,2
ByCreationDate,-2,-1,0.5
```

might be changed to different values while the rest of the protocol stays untouched.

4.3 Maintaining Collections of CEFs

The global-tuning phase of BliStrTune requires the collection *C* of CEFs as an input. It is desirable that this collection *C* is limited in size (currently we use max. 50 CEFs) and that it contains the best performing CEFs.

Initially, for each weight function *w* defined in *E*, we have extracted the CEF most often used in the *E* auto-schedule mode. We have added a CEF for each of our new weight functions. This gave us the initial collection of 21 CEFs. Then we use a global database (shared by different BliStrTune runs) in which we store all CEFs together with the usage counter which states how often each CEF was used in a protocol invented by BliStrTune. Recall that in one BliStrTune iteration, ParamILS is ran four times (two phases of global-tuning and two phases of fine-tuning). Whenever a CEF is contained in a protocol invented by any BliStrTune iteration (after the four ParamILS runs), we increase the CEF usage counter, perhaps adding a new CEF to the database when used for the first time.

To select the 50 best performing CEFs we start with $C = \emptyset$. We extract all the weight functions *W* used in the global CEF database. This set *W* stays constant because the database already contains all possible weight functions from the very beginning. For each $w \in W$, we compute the list C_w of all CEFs from the database which use *w* and sort it by the usage counter. Then we iterate over *W* and for each *w* we move the most often used CEF from C_w to *C*. We repeat this until *C* has the desirable size (or we are out CEFs). This ensures that *C* contains at least one CEF for each weight function.

5. Experimental Evaluation

This section provides an experimental evaluation¹ of BliStrTune system. In Section 5.1 we compare our improved BliStrTune with the original BliStr, and we use BliStrTune to evaluate the value added by the new weight functions. In Section 5.2 we evaluate the BliStrTune runs with different parameters. In Section 5.3 we discuss and compare several methods to construct a protocol scheduler that tries several protocols to solve a problem. Section 5.4 then compares the best protocol scheduler with state-of-the-art ATPs, namely, with *E* 1.9 using its auto-schedule mode and with Vampire 4.0.

For the evaluation we use problems from the Mizar@Turing division of the CASC 2012 (Turing100) competition mentioned in Section 1. These problems come from the MPTP

¹ All the experiments were run on 2x16 cores Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz with 128 GB memory. One prover run was however limited to 1 GB memory limit.

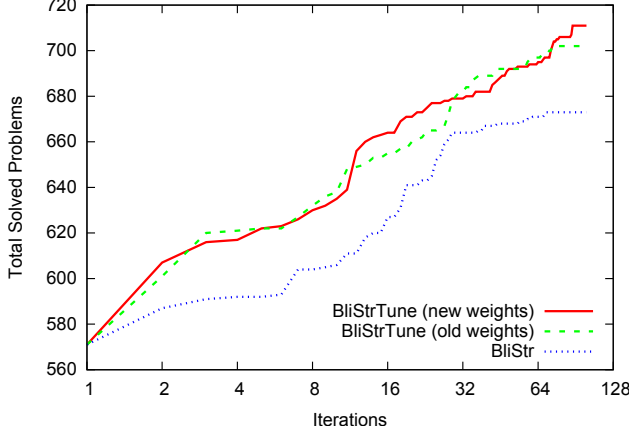


Figure 1. Value added by parameter fine-tuning and by new weight functions (Section 5.1).

translation (Urban 2004, 2006; Alama et al. 2014) of the Mizar Mathematical Library (Grabowski et al. 2010). The problems are divided into 1000 training and 400 testing problems. The training problems were published before the competition, while the testing problems were used in the competition. This fits our evaluation setting: we can use BliStrTune to invent targeted protocols for the training problems and then evaluate them on the testing problems.

5.1 Hierarchical Invention and Weight Functions

To evaluate the hierarchical invention we ran BliStr and BliStrTune with equivalent arguments. Furthermore, we ran two instances of BliStrTune to evaluate the performance added by the new weight functions from Section 2.2. The first instance was allowed to use only the original E 1.9 weight functions, while the second additionally used our new weight functions.

BliStr and BliStrTune used the same input arguments. The first argument is the set of the training problems. We use the 1000 training problems from the Mizar@Turing competition in all experiments. Other arguments are:

- $T_{improve}$ the time limit (seconds) for one ParamILS run,
- t_{cutoff} the time limit for E prover runs within ParamILS,
- t_{eval} the time limit for the protocol evaluation in BliStr/Tune.

In BliStrTune, ParamILS is run four times in each iteration, hence we set $T_{improve} = 100$ in BliStrTune and $T_{improve} = 400$ in BliStr.² We set $t_{cutoff} = 1$ and $t_{eval} = 5$ and additionally, in the case of BliStrTune, $c_{cef} = 6$.

The results are shown in Figure 1. In each iteration (x-axis, logarithmic scale) we count the total number of the training problems solved (y-axis) by all the protocols invented so far, provided each protocol is given the time

² So that the times used to improve a protocol are equal.

limit t_{eval} . This metric gives us relatively good idea of the BliStr/Tune progress.

The original BliStr solved 673 problems, BliStrTune without the new weights solved 702 problems, while BliStrTune with the new weights solved 711 problems. From this and from the figure we can see that the greatest improvement is thanks to the hierarchical parameter invention. However, the new weight functions still provide 9 more solved problems which is a useful additional improvement.

5.2 Influence of the BliStrTune Input Arguments

In this section we evaluate several BliStrTune runs with different input arguments. We run all the combinations of $T_{improve} \in \{100, 300\}$ and $c_{cef} \in \{6, 10\}$ and $t_{cutoff} \in \{1, 2\}$. This gives us 6 different BliStrTune runs. We always set $t_{eval} = 5 \cdot t_{cutoff}$.

The results are summarized in Table 1. Column *iters* contains the number of iterations executed by the appropriate BliStrTune run, *proto* is the total number of protocols generated, *run time* is the total run time of the given BliStrTune run, *best proto* is the number of training problems solved by the best protocol within t_{eval} time limit, and *solved* is the total number of the training problems solved by all the generated protocols, provided each protocol is given time limit t_{eval} . We can see that a huge amount protocols were generated. Only few of them were used for the final evaluation as described in Section 5.3. Those used for the final evaluation are considered “useful” and the column *useful* states how many percent of the useful protocols come from the appropriate BliStrTune run.

We can see that the most useful runs are the basic runs with smaller $T_{improve}$ which also have lower run times. Higher $T_{improve}$ leads to higher run times but it produces better protocols in the sense that a smaller number of protocols can solve equal number of problems. From the table we can see that when t_{cutoff} and c_{cef} are increased, $T_{improve}$ should be increased as well to provide ParamILS enough time for protocol improvement.

5.3 Selecting Best Protocol Scheduler

The 6 runs of BliStrTune described above in Section 5.2 generated more than 900 different protocols. In this section we try to select the best subset of protocols and construct a *protocol scheduler* which sequentially tries several protocols to solve a problem. We only experiment with the simplest schedulers where the time limit for solving a problem is equally distributed among all the protocols within a scheduler. Hence the problem of scheduler construction is reduced to the selection of the right protocols.

We use three different ways to select scheduler protocols. Firstly we use a *greedy* approach as follows. We evaluate all the protocols on all the training problems with a fixed time limit t . Then we construct a greedy covering sequence which starts with the best protocol, and each next protocol in the sequence is the protocol that adds most solutions to

$T_{improve}$	t_{cutoff}	t_{eval}	c_{cef}	iters	protos	run time	best proto	solved	useful
100	1	5	6	115	116	1d0h	572	711	28%
100	1	5	10	111	115	1d3h	594	715	14%
300	1	5	6	83	87	1d13h	596	698	4%
300	1	5	10	82	85	1d22h	611	711	11%
100	2	10	6	152	148	1d20h	579	720	27%
100	2	10	10	88	88	1d4h	567	698	1%
300	2	10	6	153	153	3d18h	583	727	19%
300	2	10	10	139	139	3d9h	587	719	15%

Table 1. Evaluation of different BliStrTune training runs on Mizar@Turing problems (Section 5.2).

scheduler	protos	training		testing		prover	training		testing	
		solved	V+	solved	V+		solved	V+	solved	V+
greedy ₁	33	744	+9.8%	280	+5.2%	E (BliStrTune)	744	+9.8%	280	+5.2%
greedy ₂	27	742	+9.6%	279	+4.8%	Vampire 4.0	677	+0%	266	+0%
greedy ₅	28	734	+8.4%	280	+5.2%	E (auto-schedule)	605	-10.6%	231	-13.1%
greedy ₁₀	22	719	+6.2%	276	+3.8%					
SOTAC ₁₅	15	663	-2.0%	261	-1.8%					
SOTAC ₃₀	30	693	+2.3%	266	+0%					
SOTAC ₄₅	45	698	+3.1%	270	+1.5%					
SOTAC ₆₀	60	699	+3.2%	270	+1.5%					
Σ -SOTAC ₁₅	15	692	+2.2%	268	+0.7%					
Σ -SOTAC ₃₀	30	711	+5.0%	273	+2.6%					
Σ -SOTAC ₄₅	45	712	+5.1%	276	+3.8%					
Σ -SOTAC ₆₀	60	707	+4.4%	275	+3.4%					

Table 2. BliStrTune schedulers evaluation in 60 seconds on Mizar@Turing problems (Section 5.3).

the union of problems solved by all previous protocols in the sequence. The resulting scheduler is denoted $greedy_t$.

Second way to construct a scheduler is using *state-of-the-art contribution* (SOTAC) used by CASC. A SOTAC for the problem is the inverse of the number of protocols that solved the problem. A protocol SOTAC is the average SOTAC over the problems it solves. We can sort the protocols by SOTAC and select first n protocols from this sequence. The resulting scheduler is denoted $SOTAC_n$.

SOTAC of a protocol will be high even if the protocol solves only one problem which no other protocol can solve. That is why also the Σ -SOTAC value (Kaliszyk and Urban 2014) is introduced: the sum of problem SOTAC over all the problems. This gives us schedulers denoted Σ -SOTAC _{n} .

The evaluation of 12 different schedulers with 60 seconds time limit on the training problems is provided in Table 2.

Table 3. Evaluation of the best BliStrTune scheduler on testing problems with 60 seconds time limit.

Column *protos* specifies the count of protocols within the scheduler. We shall use this evaluation to select the best scheduler, hence the results on the 400 testing problems are provided for reference only. Column *solved* is the number of problems solved in 60s. Column *V+* is a percentage gain/lost on a state-of-the-art prover Vampire 4.0 which solves 667 of the 1000 training problems and 266 of the 400 testing problems.

We can see that the best results are achieved by scheduler $greedy_1$, which also gives the best results on the testing problems. Generally, it is better to run a bigger number of protocols with lower individual time limit.

Furthermore, we can use the constructed schedulers to evaluate the contribution of our new weight functions by analyzing weight functions used in the schedulers. Table 4 summarizes the usage of different weight functions in the final schedulers. Our weight functions are referred to by their names from Section 2.2 while the original weights are called by their E prover names. Column *count* states how many times the corresponding weight function was used in some scheduler protocol, while column *freq* sums the frequencies of occurrences of CEFs which use the given weight function. We can see that our new weight function Term was the most often used weight function. Four of our weight functions were, however, not used very often which we attribute to their higher time complexity.

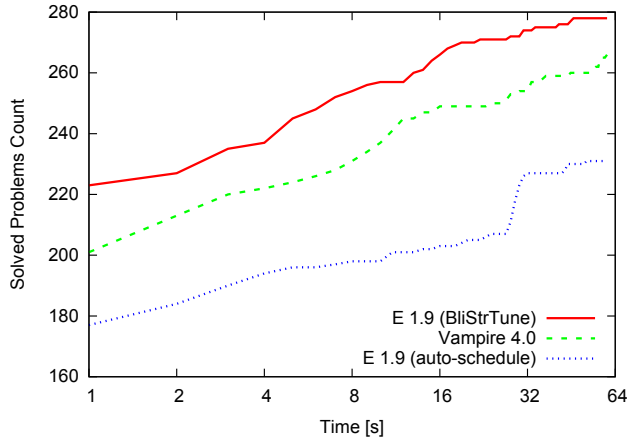


Figure 2. Progress of ATPs on the 400 Mizar@Turing testing problems with 60 seconds time limit.

5.4 Best Protocol Scheduler Evaluation

In this section we evaluate the best protocol scheduler greedy_1 selected in previous Section 5.3 on the testing problems with 60 seconds time limit. We compare greedy_1 with two state-of-the-art ATPs: (1) with E prover 1.9 in auto-schedule mode and (2) with Vampire 4.0 in CASC mode.

The results are summarized in Table 3. We can see that E with scheduler greedy_1 invented by BliStrTune outperforms Vampire by 5.2% and the improvement from E in auto-schedule mode is even more significant. Figure 2 provides a graphical representation of ATP’s progress. For each second (x-axis, logarithmic scale) we count the number of problems solved so far (y-axis). We can see that greedy_1 was outperforming Vampire during the whole evaluation.

6. Conclusions and Future Work

In this paper we have described BliStrTune, an extension of a previously published system BliStr, which can be used for hierarchical invention of protocols targeted for a given benchmark problems. The main contribution of BliStrTune is that it considers a much bigger space of protocols by interleaving the global-tuning phase with argument fine-tuning. We have evaluated the original BliStr and our BliStrTune on the same input data and experimentally proved that BliStrTune outperforms BliStr. We have evaluated several ways of creating protocol schedulers and showed that E 1.9 with the best protocol scheduler constructed from BliStrTune protocols targeted for training problems outperforms state-of-the-art ATP Vampire 4.0 on independent testing problems by more than 5%.

Furthermore, we have used BliStrTune to evaluate a contribution of our previously designed weight functions in E prover. We have shown that the new weight functions allow us to solve more problems and that (at least two of them) were often used in the best scheduler protocols. Interest-

ingly, more complex structural weights (like Lev, Ted) were not used very often in the schedulers even though our previous experiments suggested they might be very useful. We attribute this to their higher time complexity and we would like to investigate this in our future research.

Several topics are suggested for future work. We have shown that new weight functions can enhance E prover performance, hence more weight functions which consider term structure could be implemented. It seems that it will be better to design weight functions with lower time complexity, perhaps even providing approximate results (for example, some approximation of the Levenshtein distance which could be computed faster).

Another direction of our future research is to design more complex protocol schedulers. We have achieved good results with the simplest protocol schedulers where each protocol is given an equal amount of time when solving a problem. It would be interesting to design “smarter” schedulers and to see how many more problems can be solved.

Further direction of our future research are enhancements of the BliStr/Tune main loop. We could experiment with settings of various parameters, or with selection of training problems, or we could use parameter improvement methods other than ParamILS (Wang et al. 2016). Finally, we would like to make our implementation easier to use and to distribute it as a solid software package.

Acknowledgments

Supported by the ERC Consolidator grant number 649043 *AI4REASON*.

References

- J. Alama, T. Heskes, D. Kühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014. ISSN 0168-7433. doi: 10.1007/s10817-013-9286-5.
- J. Blanchette, C. Kaliszyk, L. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016a. ISSN 1972-5787. URL <http://jfr.unibo.it/article/view/4593>.
- J. C. Blanchette, D. Greenaway, C. Kaliszyk, D. Kühlwein, and J. Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016b. doi: 10.1007/s10817-016-9362-8. URL <http://dx.doi.org/10.1007/s10817-016-9362-8>.
- T. Gauthier and C. Kaliszyk. Premise selection and external provers for HOL4. In *Certified Programs and Proofs (CPP’15)*, LNCS. Springer, 2015. doi: 10.1145/2676724.2693173. URL <http://dx.doi.org/10.1145/2676724.2693173>. <http://dx.doi.org/10.1145/2676724.2693173>.
- A. Grabowski, A. Kornilowicz, and A. Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.
- F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: an automatic algorithm configuration framework. *J. Artificial Intelligence Research*, 36:267–306, October 2009.

weight function	count	freq
Term	111	782
RelevanceLevelWeight2	104	353
Pref	45	297
ConjectureGeneralSymbolWeight	43	235
FIFOWeight	39	174
StaggeredWeight	36	199
SymbolTypeweight	34	96
ConjectureRelativeSymbolWeight	28	110
ConjectureSymbolWeight	23	44
RelevanceLevelWeight	21	40
OrientLMaxWeight	13	24
Refinedweight	12	88
Defaultweight	10	131
Clauseweight	9	64
ClauseWeightAge	6	10
PNRefinedweight	5	21
Struc	5	14
Lev	3	37
Uniqweight	2	2
Ted	1	13
Tfldf	1	1
total	551	2735

Table 4. Usage of weight functions in best schedulers.

- J. Jakubův and J. Urban. Extending E prover with similarity based clause selection strategies. In *Intelligent Computer Mathematics - 9th International Conference, CICM 2016, Białystok, Poland, July 25-29, 2016, Proceedings*, pages 151–156, 2016.
- C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014. doi: 10.1007/s10817-014-9303-3. URL <http://dx.doi.org/10.1007/s10817-014-9303-3>.
- C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015. doi: 10.1007/s10817-015-9330-8. URL <http://dx.doi.org/10.1007/s10817-015-9330-8>.
- C. Kaliszyk, J. Urban, and J. Vyskocil. Efficient semantic features for automated reasoning over large theories. In Q. Yang and M. Wooldridge, editors, *IJCAI’15*, pages 3084–3090. AAAI Press, 2015a.
- C. Kaliszyk, J. Urban, and J. Vyskocil. Machine learner for automated reasoning 0.4 and 0.5. In S. Schulz, L. D. Moura, and B. Konev, editors, *PAAR-2014. 4th Workshop on Practical Aspects of Automated Reasoning*, volume 31 of *EPiC Series in Computing*, pages 60–66. EasyChair, 2015b.

- L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *CAV*, volume 8044 of *LNCs*, pages 1–35. Springer, 2013. ISBN 978-3-642-39798-1.
- D. Kühlwein and J. Urban. MaLeS: A framework for automatic tuning of automated theorem provers. *J. Autom. Reasoning*, 55(2):91–116, 2015. doi: 10.1007/s10817-015-9329-1. URL <http://dx.doi.org/10.1007/s10817-015-9329-1>.
- J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014. ISBN 978-1107077232.
- V. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
- W. McCune. Otter 2.0. In *International Conference on Automated Deduction*, pages 663–664. Springer, 1990.
- W. W. McCune. Otter 1.0 users’ guide. Technical report, Argonne National Lab., IL (USA), 1989.
- W. W. McCune. *Otter 3.0 reference manual and guide*, volume 9700. Argonne National Laboratory Argonne, IL, 1994.
- S. Schäfer and S. Schulz. Breeding theorem proving heuristics with genetic algorithms. In G. Gottlob, G. Sutcliffe, and A. Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 263–274. EasyChair, 2015. URL http://www.easychair.org/publications/paper/Breeding_Theorem_Proving_Heuristics_with_Genetic_Algorithms.
- S. Schulz. E – a brainiac theorem prover. *AI Communications*, 15(2):111–126, 2002.
- S. Schulz. System description: E 1.8. In K. L. McMillan, A. Middeldorp, and A. Voronkov, editors, *LPAR*, volume 8312 of *LNCs*, pages 735–743. Springer, 2013. ISBN 978-3-642-45220-8. doi: 10.1007/978-3-642-45221-5_49. URL http://dx.doi.org/10.1007/978-3-642-45221-5_49.
- G. Sutcliffe. The 6th IJCAR automated theorem proving system competition - CASC-J6. *AI Commun.*, 26(2):211–223, 2013. doi: 10.3233/AIC-130550. URL <http://dx.doi.org/10.3233/AIC-130550>.
- J. Urban. MPTP - Motivation, Implementation, First Experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004. doi: 10.1007/s10817-004-6245-1.
- J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.
- J. Urban. BliStr: The Blind Strategymaker. In G. Gottlob, G. Sutcliffe, and A. Voronkov, editors, *GCAI 2015. Global Conference on Artificial Intelligence*, volume 36 of *EPiC Series in Computing*, pages 312–319. EasyChair, 2015.
- J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskočil. MaLAREa SG1 - Machine Learner for Automated Reasoning with Semantic Guidance. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *LNCs*, pages 441–456. Springer, 2008. ISBN 978-3-540-71069-1.
- Z. Wang, F. Hutter, M. Zoghi, D. Matheson, and N. de Freitas. Bayesian optimization in a billion dimensions via random em-

beddings. *Journal of Artificial Intelligence Research*, 55:361–387, 2016.

K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, Dec. 1989. ISSN 0097-5397. doi: 10.1137/0218082. URL <http://dx.doi.org/10.1137/0218082>.